Highly scalable distributed-memory sparse triangular solution algorithms*

Yang Liu† Mathias Jacquelin† Pieter Ghysels† Xiaoye S. Li†

Abstract

This paper presents a highly efficient distributed-memory parallel sparse triangular solver. The triangular solution phase is often performed following factorization phase in the sparse linear solvers and has become increasingly computationally expensive for direct solvers with many right hand sides (RHSs) or preconditioned iterative solvers. However, the low arithmetic intensity and sequential nature of the triangular solve algorithm pose performance challenges for its large-scale distributed-memory parallelization. In this work, we propose several strategies to enhance scalability of an algorithm with 2D block cyclic process layout. First, an asynchronous binary-tree-based communication scheme implemented via non-blocking MPI functions is leveraged to broadcast partial solutions and reduce partial updates among a subset of processes for each block column and row of the triangular matrix, respectively. This scheme reduces message latency, improves communication load balance and significantly accelerates asynchronous execution of the triangular solve. In addition, efficient BLAS operations and threading implementations are exploited to accelerate local computations and further reduce process idle time. The proposed strategies are implemented in `SuperLU_DIST` and numerical experiments show up to 4.4x improvement with one right-hand side and up to 6.1x improvement with 50 right-hand sides on 4096 processes, compared to the current release. This is the first time that sparse triangular solution is demonstrated strong scaling on more than 4000 cores.

1 Introduction

Factorization based sparse solvers and preconditioners are indispensable methods for solving large-scale algebraic systems arising from multiphysics and multiscale simulations. Here, we focus on LU factorization $A = LU$ followed by triangular solutions with the lower and upper triangular matrices L and U. For many typical sparse matrices from 3D discretized partial differential equations with n variables, the operation cost for factorization is $O(n^2)$, and that for triangular solution is of lower order, $O(n^{4/3})$, which is proportional to the L factor size. In practice, one triangular solution often takes less than 5-10% of the factorization time. For many years, most of the research efforts for high performance direct solvers have been devoted to improving factorization (see [4, 12] and references therein). In recent years, we see more and more use of direct solvers in the preconditioning context, for linear systems or for eigen systems [1, 20, 21]. In these cases, many triangular solutions are needed using the same factorization, which often become a performance bottleneck for the overall solver. Therefore, highly efficient parallel triangular solutions are called for. However, it is more challenging to optimize triangular solution than factorization on large-scale distributed-memory machines for at least the following reasons. 1) It has relatively lower arithmetic intensity, about $O(1)$, as measured by flops per byte of DRAM access or of communication. As a result, the parallel computation is strictly communication dominant, which gives little chance of overlapping computation with communication. Reducing communication costs is critical for improving the overall triangular solve performance, especially on large-scale high performance computers. 2) It has stronger task dependencies resulting from the sparsity pattern of the factorized triangular matrices L and U. Such task dependency sequentializes the solve algorithm and introduces parallelization difficulties on distributed-memory machines. Thus, better task scheduling and faster task execution is critical to reduce synchronization and process idle time.

To the best of our knowledge, most existing parallel triangular solvers are targeting shared-memory machines or GPU (see [2, 16, 17, 25] and references, therein). These solvers often rely on well-known techniques such as the level-set, color-set, or block scheduling algorithms. However, the high inter-node communication costs associated with these techniques limit their efficiency on distributed-memory machines. Consequently, existing distributed-memory parallel triangular solvers generally exhibit limited strong scalability. To name a few, a multifrontal triangular solver that distributes frontal matrices among processes was reported in [19]. A non-blocked triangular solver with a 1D process layout and improved computation load balance was developed in [24]. Our previous supernodal triangu-

*This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

†Scalable Solvers Group, Lawrence Berkeley National Laboratory, (liuyangzhuan|mjacquelin|pghysels|xsli@lbl.gov)

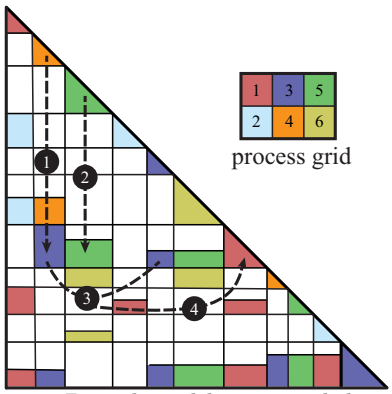


Figure 1: Distributed layout and data flow.

lar solver in the SuperLU_DIST software [13, 14] relies on 2D cyclic process layout to achieve improved computation load balance. All these solvers show limited scalability for large process counts (e.g., thousands of processes), due to the aforementioned difficulties. Note that it is also possible to factorize the inverse of triangular matrix as multiple sparse factors to exploit more parallelism across factors [5, 18]. In addition, a communication-avoiding parallel algorithm is proposed for dense triangular systems with multiple right hand sides (RHSs) [26].

In this work we introduce several techniques to address the above issues. Although we implement these techniques in the SuperLU_DIST software, the ideas are transferable to any other triangular solution software that uses 2D block cyclic layout. The motivation is as follows: the high communication cost is largely due to the need to broadcast and reduce partial solutions among processes along each block column and row, respectively. It is possible to leverage standard collective MPI functions, e.g., MPI_Bcast/MPI_Reduce, to reduce message latency and balance communication load in each collective operation. Unfortunately, these functions cause high memory/communication overheads due to their construction/synchronization. Here, we exploit the binary-tree communication models implemented in standard MPI functions to develop a lightweight communication algorithm. As these operations require asynchronous communications among a different subset of processes per column/row, we efficiently construct a binary communication tree per process group and leverage non-blocking MPI functions to propagate the messages asynchronously across different trees. An optimized message priority condition is enforced inside and among communication trees to enhance pipelining of the computations. In addition, more efficient and threaded implementations of local computation are developed to further reduce process idle time. Our numerical experiments show these techniques permit significant

Let $PROC_C(K)/PROC_R(K)$ be processes for column/row K
1. Initialization: $x := b$; $lsum := 0$
   /* —- Process leaf nodes —- */
2. **for** block $K = 1$ to N
3. **if** (I am diagonal process owning $x(K)$
4. **and** $fmod(K) = 0$)
5. $x(K) = L(K,K)^{-1}x(K)$
6. Send $x(K)$ to the column processes $PROC_C(K)$
7. **endif**
8. **end for**

   /* —- Process internal nodes —- */
9. **while** (I expect more arriving messages) **do**
10. Receive a message
11. **if** (message is $x(K)$)
12. **for** (each of my $L(I,K) \neq 0, I > K$)
13. $lsum(I) = lsum(I) + L(I,K)x(K)$
14. $fmod(I) = fmod(I) - 1$
15. **if** ($fmod(I) = 0$)
16. Send $lsum(I)$ to the diagonal process owning $x(I)$
17. **endif**
18. **end for**
19. **else if** (message is $lsum(K)$)
20. $x(K) = x(K) - lsum(K)$;
21. $fmod(K) = fmod(K) - 1$
22. **if** ($fmod(K) = 0$)
23. $x(K) = L(K,K)^{-1}x(K)$
24. Send $x(K)$ to the column processes $PROC_C(K)$
25. **endif**
26. **endif**
27. **end while**

Algorithm 1: Parallel lower triangular solve $Lx = b$.

scalability improvement across varieties of benchmark matrices and large range of process counts.

The rest of the paper is organized as follows. Section 2 briefly describes the sequential triangular solve algorithm in SuperLU_DIST and our baseline parallel implementation [14]. Next, improved algorithms that leverage binary-tree-based communication models and efficient local computations are elucidated in Section 3. Numerical results that demonstrate the superiority of the proposed techniques are presented in Section 4.

2 Triangular solution and its baseline parallelization

Consider the solution of a $n \times n$ linear system $Lx = b$, where L is a lower triangular matrix, and b is a $n \times k$ right-hand side (RHS) matrix or vector (k=1). The i-th solution row of x is computed by recurrence

$$x_i = (b_i - \sum_{j=1}^{i-1} L_{ij} \cdot x_j)/L_{ii} \ . \tag{2.1}$$

For a sparse matrix L, computation of x_i needs some or all of the previous solution rows $x_j, j < i$, depending on the sparsity pattern of the i-th row of L. This computation dependency can be precisely modeled

by the directed acyclic graph (DAG) $G(L^T)$ as follows: there are n vertices corresponding to n variables/rows $x_i, i = 1 \ldots, n$. There is a directed edge from x_j to x_i if $L_{ij} \neq 0$. In a blocked algorithm, such as supernodal or multifrontal algorithms, we work with a coarser supernodal DAG where each vertex represents a group of x_i variables and an edge represents the dependency between two groups of x_i variables. In this work, we always use supernodal representation. Note that the proposed methodologies also apply to other non-blocked and blocked algorithms. Let $b(K)$ and $x(K)$ denote the subvector corresponding to supernode K and $L(I, K)$ denote the nonzero submatrix corresponding to supdernodes I and K. The solution subvector $x(K)$ is computed as

$$x(K) = L(K,K)^{-1}\Big(b(K) - \sum_{I=1}^{K-1} L(K,I) \cdot x(I)\Big) \ . \tag{2.2}$$

The distributed-memory parallel algorithm partitions the matrix L among multiple processes, and each process is in charge of a subset of solution subvectors $x(K)$. The solution subvectors and partial summation results in (2.2) are communicated during the solve phase. Note that for simplicity we do not consider distributing the RHS b along its column dimension when the number of RHS $k > 1$. Due to low arithmetic intensity and high sequentiality, a process can stay mostly idle when the process count becomes large. In what follows, we propose several techniques to alleviate these scaling hurdles.

Our starting point of this work is based on the earlier parallel algorithm in `SuperLU_DIST`, which first appeared in [13], and had little update since then [14]. We recall that `SuperLU_DIST` uses a 2D block-cyclic distribution for the factored L and U matrices. Figure 1 illustrates such a distribution and associated communication requirements with a 2×3 process grid. Processes owning the diagonal blocks (called *diagonal processes*) are responsible for computing the corresponding blocks of the x components. When $x(I)$ is needed in $L(K, I)x(I)$, and the owners of $x(I)$ and $L(K, I)$ are different, $x(I)$'s process needs to send it to the process of $L(K, I)$, see ① in Figure 1. Note that $L(I, I)$ and $x(I)$ are owned by the same process. In case of ②, no communication is needed because both $x(I)$ and $L(K, I)$ reside on the same process 5 (marked in green). After receiving required $x(I)$ entries, each process proceeds with local summation, i.e., step ③ in Figure 1. Finally, the local sums are sent to the diagonal process which performs the inversion, see ④ in the figure.

For completeness, we list the pseudo-code for the lower triangular solve in Algorithm 1 and describe the details here. For illustration purposes, it is assumed

Matrix	Comp	Tot	Tot (no DAG)
cage13 (1 RHS)	1.4E-01	3.7E-01	2.1E-01
Li4244 (50 RHSs)	1.8E-01	3.7E-01	1.9E-01

(a)

Matrix	Comp	Tot	Tot (no DAG)
cage13 (1 RHS)	6.0E-03	2.3E-01	1.4E-01
Li4244 (50 RHSs)	1.1E-02	1.6E-01	3.6E-02

(b)

Table 1: Time (in seconds) of triangular solve in `SuperLU_DIST` with (a) 8x8 and (b) 45x45 grids. The computation time is the maximum over all processes

that a diagonal process owns no off-diagonal blocks on the same supernode row. Let $PROC_C(K)/PROC_R(K)$ denote the set of processes along block column/row K. In this formulation, before the K-th subvector $x(K)$ is solved, the update from the inner product of $L(K, 1 : K-1)$ and $x(1 : K-1)$ must be accumulated in $lsum(K)$ and subtracted from $b(K)$. The diagonal process is responsible for solving $x(K)$. During the setup phase, the dependency graph allows us to compute a counter $fmod(K)$ on the processes residing along block row K. The counter counts how many local blocks $L(I, K)$ will contribute to $lsum(K)$ `and` how many $lsum(K)$ updates the owner of $x(K)$ will receive from a subset of the non-diagonal processes along the block row K. During the solve phase, $fmod(K)$ is updated to keep track of when dependency is met, enabling asynchronous executions. Specifically, the counter is decremented whenever a local contribution to $lsum(K)$ is performed or an nonlocal $lsum(K)$ is received. Once $fmod(K)$ reaches zero, the non-diagonal process sends $lsum(K)$ to the diagonal process owning $x(K)$; the diagonal process solves $x(K)$ and sends it to a subset of $PROC_C(K)$.

The execution of the program is *message-driven*. A process may receive two types of messages, one is the partial sum $lsum(K)$, another is the solution subvector $x(K)$. Appropriate action is taken according to the message type. Note that the messages are communicated via MPI_Isend and MPI_Recv. The asynchronous communication enables certain degree of overlap between communication and computation, and helps reduce process idle time. This is very important for such communication-dominant applications.

To assess performance of Algorithm 1, we consider the `SuperLU_DIST` factored L matrices of two benchmark matrices with different densities (defined in Table 2 caption), cage13 and Li4244 (see Table 2). Matrix cage13 with density 2.36% is run with $k = 1$ RHS; matrix Li4244 with density 19.2% is run with $k = 50$ RHSs. When executed with a 8×8 process grid (see Table 1a), the computation time (corresponding to lines 5, 13 and 23 of Algorithm 1) of both runs are comparable to the

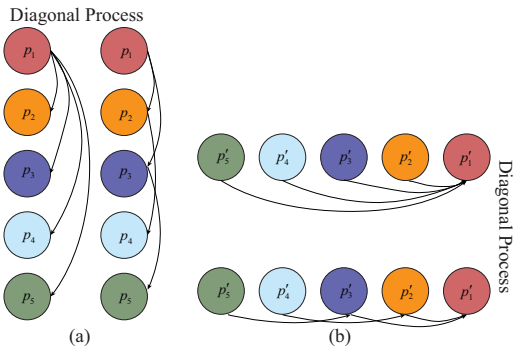


Figure 2: Comparison between flat and binary trees: (a) broadcast tree, (b) reduction tree.

communication and idle time. However when executed with a 45×45 process grid (see Table 1b), more than 90% of the total time is spent in communication and waiting. This is due to the low arithmetic intensity per message (note that matrix Li4244 has higher density and hence higher arithmetic intensity than cage13). In addition to low arithmetic intensity, another bottleneck is the sequentiality associated with the sparsity pattern. To illustrate the effect of sequentiality, both matrices are run with $fmod(K) = 0$ for all K to remove the DAG dependency while keeping the same amount of communication, thus ignoring the correctness of the solution (see last columns of the two tables). When removing dependencies, the runs become entirely computation dominant with the 8×8 process grid but still communication dominant with the 45×45 process grid.

It is also worth-mentioning that the solve phase becomes comparable to or dominant over factorization phase in direct solvers with many RHSs or preconditioned iterative solvers. Take the example of cage13 with the 45×45 process grid, factorization and solve (per RHS) require 59.25s and 0.23s, respectively. This clearly demonstrate the need for better triangular solve algorithms.

In the following section, we propose several strategies including efficient communication algorithms and faster local computation to reduce communication latencies and process idle time.

3 Improved Algorithms

3.1 Asynchronous binary-tree-based collective communications. As discussed, Algorithm 1 involves an excessive amount of communication but little computation. Recall that there are two types of communications along the column and row dimensions. Whenever a subvector $x(K)$ is solved, its diagonal process sends it to the column processes $PROC_C(K)$ that own nonzero blocks in supernode column K (lines 6 and 24 in Algorithm 1). Similarly, whenever a local sum

Let $BT(K)/RT(I)$ be the broadcast/reduction process tree for column/row K
1. Initialization: $x := b$; $lsum := 0$
   /* —- Process leaf nodes —- */
2. **for** block $K = 1$ to N
3. **if** (I am diagonal process owning $x(K)$
4. **and** $fmod(K) = 0$)
5. $x(K) = L(K,K)^{-1}x(K)$
6. Forward $x(K)$ in broadcast tree $BT(K)$
7. **endif**
8. **end for**

   /* —- Process internal nodes —- */
9. **while** (I expect more arriving messages) **do**
10. Receive a message
11. **if** (message is $x(K)$)
12. Forward $x(K)$ in broadcast tree $BT(K)$
13. **for** each of my $L(I,K) \neq 0, I > K$
14. $lsum(I) = lsum(I) + L(I,K)x(K)$
15. **end for**
16. **for** each of my $L(I,K) \neq 0, I > K$
17. $fmod(I) = fmod(I) - 1$
18. **if** ($fmod(I) = 0$)
19. Forward $lsum(I)$ in reduction tree $RT(I)$
20. **endif**
21. **end for**
22. **else if** (message is $lsum(K)$)
23. **if** (I am diagonal process owning $x(K)$)
24. $x(K) = x(K) - lsum(K)$
25. $fmod(K) = fmod(K) - 1$
26. **if** ($fmod(K) = 0$)
27. $x(K) = L(K,K)^{-1}x(K)$
28. Forward $x(K)$ in broadcast tree $BT(K)$
29. **endif**
30. **else**
31. add message to local $lsum(K)$
32. $fmod(K) = fmod(K) - 1$
33. **if** ($fmod(K) = 0$)
34. Forward $lsum(K)$ in reduction tree $RT(K)$
35. **endif**
36. **endif**
37. **endif**
38. **end while**

Algorithm 2: Binary-tree-enhanced parallel lower triangular solve $Lx = b$.

$lsum(I)$ finishes accumulation, its off-diagonal process sends it to the diagonal process owning $x(I)$ (line 16 in Algorithm 1). The diagonal process receives local sums from a subset of row processes $PROC_R(I)$ that own nonzero blocks in supernode row I. Note that these communications are collective in nature: the column-wise one is a broadcast operation, the row-wise one is a reduction operation. These communications are implemented with MPI_Isend and MPI_Recv. (Note that we do not use non-blocking MPI_Irecv as there is little room for overlapping message receiving with computations). The implementation in Algorithm 1 is essentially based on a flat-tree communication model. It is thus clear that a single such operation involving p pro-

Matrix	Size n	Nonzeros	Nonzeros in L	Density	Description
nlpkkt80	1,062,400	28,704,672	1,641,731,498	0.29%	Optimization problem, Symmetric indefinite KKT matrix
cage13	445,315	7,479,343	2,338,129,041	2.36%	DNA electrophoresis, 13 monomers in polymer
Geo_1438	1,437,960	63,156,690	2,426,672,664	0.23%	Structural problem, Geomechanical model of earth crust
atmosmodj	1,270,432	8,814,880	968,674,795	0.12%	Fluid dynamics, Atmospheric modeling
StocF-1465	1,465,137	21,005,389	1,054,880,448	0.09%	Fluid dynamics, Flow in porous medium
A22	519,552	127,135,918	1,782,722,012	1.32%	Magnetohydrodynamics, Magnetized toroidal plasma
tdr455k	2,738,556	112,756,352	1,328,776,490	0.04%	Electromagnetics, Eigenmode for accelerator cavity
LU_C_BN_C	263,328	190,859,344	1,785,728,803	5.15%	Electronic structure theory, C-BN sheet with 20256 atoms
Ga19As19H42	133,123	8,884,839	810,605,750	9.15%	Quantum chemistry theory, Real-space pseudopotential method
Li4244	72,000	258,880,000	498,596,000	19.2%	Electronic structure theory, 3D lithium system with 4244 atoms
DG_Graphene	327,680	238,668,800	966,000,640	1.8%	Electronic structure theory, Graphene with 8192 atoms
DNA_715	459,712	224,055,744	425,579,006	0.4%	Electronic structure theory, DNA molecule with 45760 atoms

Table 2: Test matrices. Density := {nonzeros in L}$\times 2$ / n^2

cesses will yield a communication latency of p and the communication load is imbalanced across each process group (the root/diagonal process communicates excessively more messages than non-diagonal processes). For example, suppose a process requires a certain message in a broadcast operation to activate a leaf supernode in the reduced DAG. If the process is the last receiver from the root, it is more likely that the following computations will be blocked. Moreover in practice, any network has variation in the communication bandwidth and latency among different nodes. If the diagonal processes have lower bandwidth or higher latency than others, the communication performance will be further degraded.

Therefore it is beneficial to reshape the message routing patterns based on so-called binary-tree communication models. Binary-tree communication models only require $\log_2 p$ latency and balance the communication loads among all p processes. A natural way to leverage these communication models is to use MPI_Bcast and MPI_Reduce available in standard MPI libraries. However, such option is infeasible for the following reasons: 1) The MPI_Bcast and MPI_Reduce operations are blocking in nature, which degrades the performance of the asynchronous algorithm in Figure 1 particularly when one process is involved in multiple broadcast and reduction operations. Since the diagonal process sends $x(K)$ only after it is solved and a non-diagonal process sends $lsum(I)$ only when local accumulation finishes, the collective communication algorithm should also be asynchronous in nature. 2) Note that each collective operation very likely involves a different subset of process group $PROC_C(K)$ or $PROC_R(K)$. Even two collective operations involve the same subset of process group, each process can take charge of different supernode columns/rows. This requires construction of one communication group per supernode column/row either in the setup or solve phase, which turns out to be memory or computation inefficient. Therefore, even asynchronous MPI_Ibcast and MPI_Ireduce operations cannot be used. 3) The order of processes in the tree constructed by standard MPI libraries may not be optimal for pipelining the triangular solve algorithm. It is worth-mentioning that similar observations have been reported for the matrix factorization phase in [6, 22].

The aforementioned drawbacks of flat-tree and standard binary-tree communication algorithms motivate us to develop a low-overhead asynchronous tree-based communication algorithm specially tailored for the triangular solve algorithm. In what follows, we present the flat-tree and the binary-tree communication models and describe the modified triangular solve algorithm in detail.

Consider one broadcast operation as depicted in Figure 2(a) where the diagonal process p_1 sends the solved subvector $x(K)$ to a subset of $p-1$ processes in $PROC_C(K)$. In the flat-tree model as implemented in Algorithm 1, the diagonal process sends $x(K)$ one-by-one for $p-1$ times. In the binary-tree model, in contrast, each process receives at most one message from its parent and forwards at most two messages to its children. This clearly reduces the latency of the broadcast operation from $p-1$ to $\log_2 p$ and reduces maximum com-

munication volume per process from $O(p)$ to $O(1)$. On the other hand, the reduction operation in Figure 2(b) is more subtle. In the flat-tree model of Algorithm 1, the diagonal process receives $p-1$ messages one-by-one once they become ready-to-send (when local accumulation finishes) on non-diagonal processes. In contrast, each process in the binary-tree model receives at most two messages from its children and forwards at most one message (when local accumulation finishes and all children messages have been received) to its parent. By reducing the number of messages communicated with the root from $p-1$ to two, the binary-tree model efficiently reduces the chance of instantaneous hotspots in the presence of network bandwidth and latency variations and facilitates pipelining communication and computation tasks across multiple trees. Also, it is worthmentioning that the ordering of processes in a broadcast/reduction tree plays a critical role in reducing the overall communication cost. For example in a broadcast tree $BT(K)$, each participating process owns multiple blocks in the supernode column K. It is assumed that blocks with smaller supernode row numbers are likely to activate the row as leaf nodes earlier in the DAG. Therefore these blocks should have higher priority to receive the message so as to start computation or forwarding earlier. In our proposed models, processes in a broadcast tree are ordered such that a process owning smaller block row numbers appears closer to the root. For similar reasons, processes owning larger block column numbers appear closer to the root in a reduction tree. Note that one can also attempt to construct the trees with random process orders as proposed in a selected inversion code [6], however, our study shows that tree models with carefully ordered processes work better for the triangular solve algorithm.

To implement these models efficiently, we construct one broadcast tree $BT(K)$ per supernode column K out of $PROC_C(K)$ processes and one reduction tree $RT(K)$ per supernode row K out of $PROC_R(K)$ processes during the setup phase. Note that each process in a tree only keeps track of its parent and children. Such construction consumes only negligible CPU and memory resources. As in the flat-tree-based algorithm, a counter $fmod(K)$ is computed per supernode row K to record the number of local inner-products and non-local messages for their contribution to $lsum(K)$. The number of local updates equals the number of blocks in row K one process owns; the number of non-local updates equals the number of children in reduction tree $RT(K)$. As opposed to the flat-tree algorithm that expects much higher $fmod(K)$ on diagonal processes than non-diagonal ones, the binary-tree algorithm expects similar $fmod(K)$ values on all processes.

The solve phase of the modified triangular solve algorithm is outlined in Algorithm 2. Just like the flat-tree-based algorithm, the leaf supernode rows are processed first. Once solved, the root process in $BT(K)$ forwards $x(K)$ to its children via non-blocking MPI_Isend. The rest of the algorithm is message-driven. If a received message is $x(K)$, the process forwards the message in broadcast tree $BT(K)$ (line 12 in Algorithm 2) before performing local sum accumulation. Otherwise if the message is $lsum(K)$, the process accumulates it to local sums. Once $fmod(K)$ becomes zero, non-diagonal processes forward $lsum(K)$ in reduction tree $RT(K)$ (lines 19 and 34 in Algorithm 2); the diagonal process solves $x(K)$ and forwards the results in broadcast tree $BT(K)$ (line 28 in Algorithm 2).

Recall that our assumption that blocks with smaller row number are likely to activate leaf node rows earlier, local sum messages with smaller row numbers should have higher message forwarding priority given that there are multiple messages ready-to-send. Therefore, we reorder the loop at lines 16-21 such that $lsum(I)$ with smaller I is sent first. Indeed this also helps pipelining the overall algorithm.

The integration of light-weight non-blocking collective communication schemes into Algorithm 2 facilitates its asynchronous execution. By balancing communication volume per process and improving message latency, the process idle time is greatly reduced (line 10 in Algorithm 2). Recall our earlier discussions in Section 2, the lack of parallelism due to the DAG structure also poses unique challenges to minimize process idle time. To this end, one process should perform local computation faster and start message forwarding earlier to facilitate the traversal of the DAG. In addition to the tree-based communication model that improves the communication cost, several techniques are proposed next to further alleviate the lack of parallelism.

3.2 Efficient local computations. Recall that the computation tasks consist of matrix-vector multiplications involving $L(I,K)$ (line 14 of Algorithm 2) and triangular solve involving $L(K,K)$ (lines 5 and 27 of Algorithm 2). Notice that under 2D block cyclic layout, the matrix-vector multiplication becomes more computationally intensive than the triangular solve. To accelerate this computation task, the $L(I,K)$, $I > K$ are grouped into a tall-and-skinny block that leverages fast (e.g. Intel MKL) GEMV ($k = 1$) or GEMM ($k > 1$) BLAS operations (lines 13-15). Moreover, the triangular solve is also implemented as GEMV/GEMM via pre-computed $L^{-1}(K,K)$ to achieve better flop rate.

We also leverage hybrid OpenMP/MPI programming models to further improve the computation and

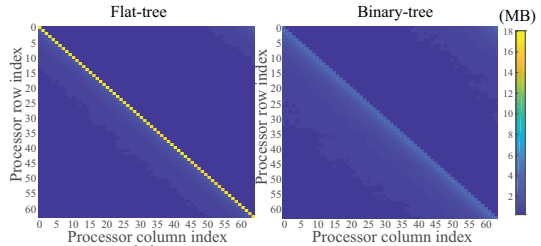
communication efficiency. The motivation is two-fold: first, faster local computation, as already discussed, enables earlier message forwarding; second, low-latency intra-node communication can be reduced via launching fewer MPI processes per node. In what follows, we briefly describe the threading component of the triangular solve in the context of Algorithm 2. We exploit two types of node-level concurrency, the first is the loop (lines 2-8) that processes all local leaf supernode rows; the second is the loop (lines 13-15) that performs GEMV operations for local sum accumulation. Note that during processing of the leaf nodes, a process will likely incur GEMV operations similar to lines 13-15 after solving one $x(K)$ and generate more leaf nodes recursively without communication. We employ a task-based threading model that hybridizes parallelization of the aforementioned loops. One OMP task is generated for several initial leaf nodes and each new leaf node. All threads update the counter $fmod(K)$ atomically without synchronization. This threading scheme is essentially an asynchronous level scheduling based scheme. In addition, the GEMV operations are parallelized among threads (as OMP tasks) once the merged block dimension is large enough. Note that the block dimension oftentimes becomes larger when the solve phase approaches its end. When executed with one MPI process, i.e., pure multi-threading, the proposed algorithm achieves similar threading performance than that of the hybrid level-scheduling and blocking algorithm proposed in [2]. When using more MPI processes, the algorithm leverages both MPI and OpenMP parallelization consistently. Although it is expected that the threading performance will degrade with large numbers of MPI processes, it should still improve computation time and thus enable earlier message forwarding.

4 Numerical Results

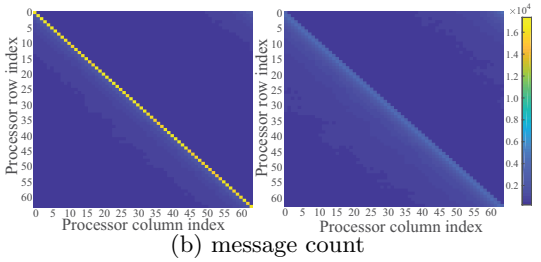
In this section, we present several numerical experiments to demonstrate the scalability of the proposed communication efficient triangular solve algorithm. The experiments are organized into three groups. First, the enhanced scalability of binary-tree-based communication algorithms is demonstrated. Second, the improvement in the local computation is validated. Finally, we present a set of benchmark matrices to test the overall algorithm.

We selected 12 benchmark matrices arising from various application domains as listed in Table 2. Several matrices are coming from collaborative research projects across U.S. Department of Energy laboratories and research institutes. Matrix A22 is generated from the M3D-C1 software [7] for magnetohydrodynamics modeling of plasma fusion. Matrix tdr455k is generated from the Omega3P software [11] for electromagnetic eigenmode analysis of accelerator cavities. Matrices LU_C_BN_C, Li4244, DG_Graphene, and DNA_715 are coming from practical large scale electronic structure calculations. More precisely, these matrices are generated from SIESTA [23] and DGDFT [15], two software packages for performing Kohn-Sham density functional theory [10] calculations using two different types of basis sets. All other matrices are publicly available through the SuiteSparse Matrix Collection [3], a widely used benchmark set of problems for testing sparse direct methods.

The benchmark matrices are first factorized via `SuperLU_DIST` with METIS ordering for fill-in reduction [8]. The L-factors are used with the proposed

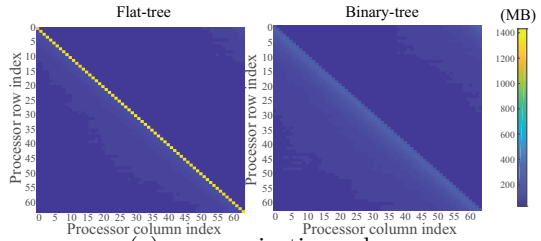


(a) communication volume

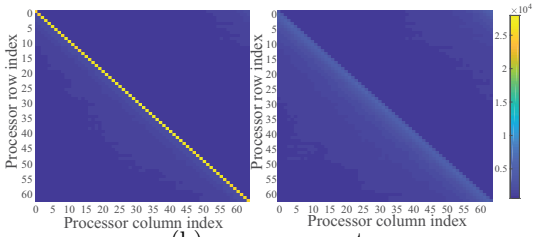


(b) message count

Figure 3: Communication heat map of tdr455k matrix with a 64×64 process grid. $k = 1$ RHS



(a) communication volume



(b) message count

Figure 4: Communication heat map of atmosmodj matrix with a 64×64 process grid. $k = 50$ RHSs

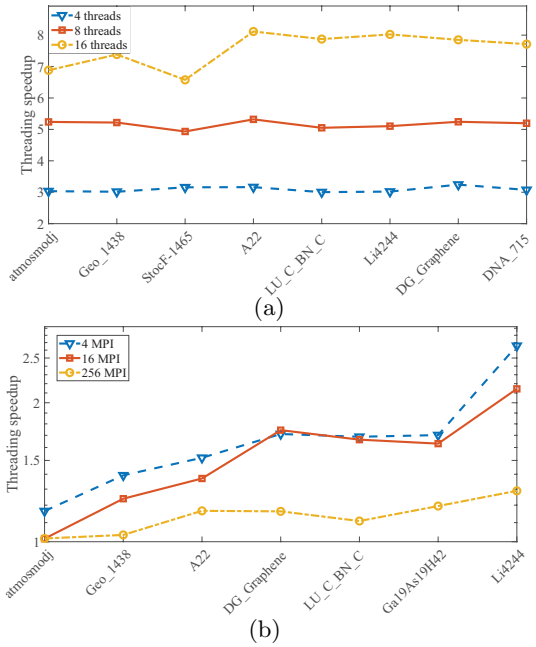


Figure 5: Threading performance w.r.t. single-threaded implementations with same number of MPI processes. (a) 1 MPI process with up to 16 threads per process. (b) 4, 16, 256 processes with 4 threads per process

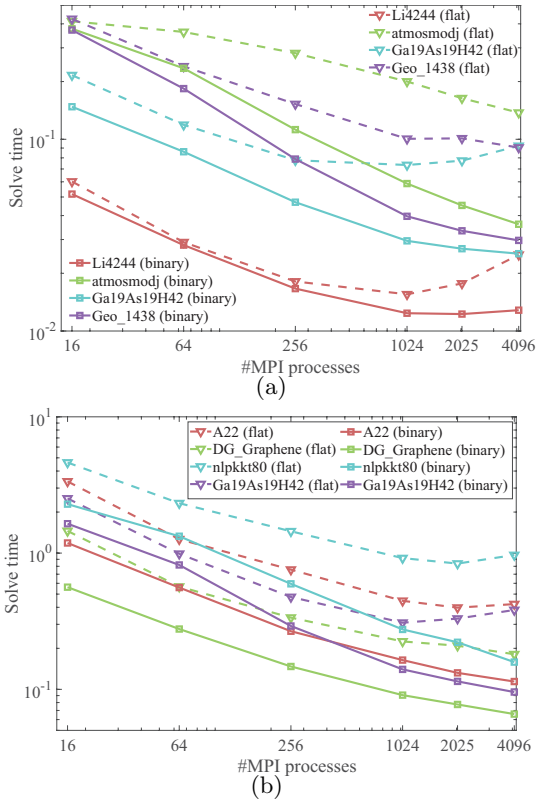


Figure 6: Scaling results for (a) 1 RHS and (b) 50 RHSs

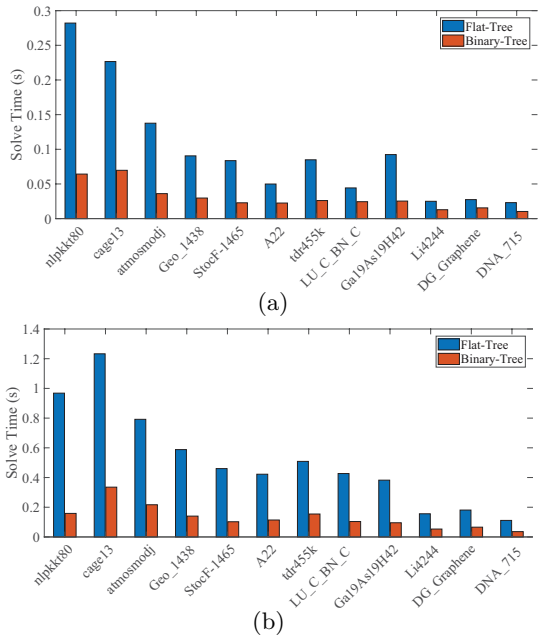


Figure 7: Solve time with 4096 MPI processes for (a) $k = 1$ RHS and (b) $k = 50$ RHSs

triangular solve algorithms. We choose matrices with different dimensions and densities to demonstrate the efficiency of the proposed algorithm. Note that the algorithm also applies to the U factors with minor modifications.

All experiments are performed on the "Cori Haswell" system at NERSC. Cori Haswell is a Cray XC40 system and consists of 2388 dual-socket nodes with Intel Xeon E5-2698v3 processors running 16 cores per socket. The nodes are configured without Hyper-Threading, run at frequency of 2.3 GHz, and are equipped with 128 GB of DDR4 memory at 2133 MHz. The nodes are connected through the Cray Aries interconnect with Dragonfly topology [9]. Unless otherwise stated, all experiments use one thread per MPI process.

4.1 Binary-tree algorithm. In the first set of numerical experiments, the communication load balance of the parallel triangular solve algorithms is analyzed using the flat-tree and binary-tree models. The algorithms are first applied to matrix tdr455k with dimension $2,738,556$ and L-factor density 0.035% using a 64×64 process grid. The supernode sizes range from 20 to 128. The triangular system is solved with $k = 1$ RHS. Figure 3 depicts the communication volume (in MB) and message count heat maps for the flat-tree and binary-tree communication models. For the flat-tree model, the communication volume and message counts are highly imbalanced. The maximum message volume and message count along diagonal processes are 18 MB and $17,355$. It is easy to see that the imbalance will

deteriorate as the process count increases. This imbalance causes network contention close to diagonal processes and leads to higher message latencies. In stark contrast, the heat maps for the proposed binary-tree model are well-balanced with maximum message volume 4.3 MB and message count 4,186. As a result, the improved communication model reduces the solve time from 84 ms to 26 ms.

Similarly, the algorithm is applied to matrix atmosmodj with dimension 1,270,432 and L-factor density of 0.12% for $k = 50$ RHSs using a 64×64 process grid. With 50 RHSs, each message size becomes much larger than with 1 RHS and can incur different communication protocols at the MPI level. As can be seen from Figure 4, the proposed communication model reduces the maximum volume and message count from 1.43 GB and 27,964 to 280 MB and 5464. Consequently, the solve time is reduced from 0.8 s to 0.31 s.

The above numerical examples demonstrate the effectiveness of binary-tree communication models in balancing communication loads and reducing message latencies for different message sizes with large process counts. It is also worth mentioning that the improvement using the binary-tree model over the flat-tree is less significant when process count is relatively small (e.g., less than 32). This is because flat-tree-based intra-memory communications benefit more from low memory latency and better cache reuse. Therefore in the proposed algorithm, binary-tree models are enabled when process count in a supernode row/column is larger than, say, core count per socket.

4.2 Faster computation. In this subsection, the effects of more efficient local computations are demonstrated for the binary-tree-based parallel triangular solve algorithm. First the locally improved algorithm is tested with a single MPI process. We observed that upon grouping matrix-vector products and precomputing inverses of diagonal blocks, the algorithm achieves 2-3x speedups for $k = 1$ RHS and 1.2-1.4x speedups for $k = 50$ RHSs. Building upon these improvements, we now focus our experiment on threading enhancement of local computation. Figure 5a depicts the threading speedups with 4, 8, and 16 threads for different benchmark matrices with $k = 1$ RHS. Irrespective of sparsity and dimension of the matrices, our algorithm can achieve up to 8x speedups with 16 threads.

Next, we report the performance of the hybrid OMP/MPI implementation of triangular solve. Figure 5b shows threading speedups with 4 threads per MPI processes with fixed process grids. As MPI process count increases, the threading performance starts to degrade. This is due to a decreased degree of both leaf-node concurrency (lines 2-8 of Algorithm 2) and local sum concurrency (lines 13-15 of Algorithm 2), i.e., decreased computation-to-communication ratios. That said, the proposed algorithm can still achieve 1.3x threading speedups with 256 MPI processes. Also note that matrices are sorted roughly based on their nonzero ratios. As the matrices become denser and exhibit higher computation-to-communication ratios, the algorithm exploits more local sum parallelism (lines 13-15 of Algorithm 2) and hence achieves better speedups.

4.3 Overall performance. In the last set of experiments, we compare the overall performance of Algorithm 2 and Algorithm 1 for matrices with varying dimensions and sparsity patterns. First, we plot the strong scaling results for $k = 1$ RHS in Figure 6a and $k = 50$ RHSs in Figure 6b. Each data point is generated by averaging multiple run results of the same matrix and CPU configuration to take account of network bandwidth and latency variations. One thread is launched per process. For many matrices, e.g., Geo_1438, Ga19As19H42, Li4244, Algorithm 1 stops scaling beyond 1,024 processes. In contrast, Algorithm 2 is able to scale to 4,096 processes. For all test matrices, Algorithm 2 achieves better performance than Algorithm 1 for the same number of processes. When the process count is small, the proposed algorithm allows more efficient local computations to achieve significant speedups; when the process count is large (for instance, beyond 256), the proposed algorithm relies more on binary-tree-based communication models to achieve better scaling performance.

Finally, we report the performance improvement of Algorithm 2 over Algorithm 1 for all benchmark matrices in Table 2 on a 64×64 process grid. Algorithm 2 achieves 1.7-4.4 fold speedups for $k = 1$ RHS (Fig. 7a) and 2.7-6.1 fold speedups for $k = 50$ RHSs (Fig. 7b), respectively.

5 Conclusion

This paper presents several techniques to improve scalability of distributed-memory parallel triangular solve algorithms. A binary-tree-based communication model with standard MPI implementations is proposed to reduce message latency and improve load balance for communicating partial solutions. In addition, the algorithm leverages efficient BLAS operations and threading accelerations to achieve faster local computations. These strategies alleviate high communication costs and reduce process idle time that plague all parallel triangular solve algorithms. We observed up to 6 fold speedups with 4,096 processes in solving lower triangular systems generated by the `SuperLU_DIST` supernodal pack-

age, which clearly demonstrates superiority of the proposed strategies compared to our baseline implementation.

The proposed algorithms can be applied to both lower and upper triangular systems arising in any supernodal, multifrontal or nonblocked triangular solve algorithms. The improved parallel efficiency suggests its applicability in accelerating preconditioned iterative and direct solvers with both single and multiple RHSs for large-scale algebraic systems.

References

[1] M. Benzi, *Preconditioning techniques for large linear systems: a survey*, J. Comput. Phys., 182 (2002), pp. 418 – 477.

[2] A. M. Bradley, *A hybrid multithreaded direct sparse triangular solver*, in Proceedings of SIAM Workshop on Combinatorial Scientific Computing, 2016, pp. 13–22.

[3] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Softw., 38 (2011), pp. 1:1–1:25.

[4] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, *A survey of direct methods for sparse linear systems*, Acta Numerica, 25 (2016), p. 383–566.

[5] N. J. Higham and A. Pothen, *Stability of the partitioned inverse method for parallel solution of sparse triangular systems*, SIAM J Sci Comput., 15 (1994), pp. 139–148.

[6] M. Jacquelin, L. Lin, N. Wichmann, and C. Yang, *Enhancing scalability and load balancing of parallel selected inversion via tree-based asynchronous communication*, in IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, pp. 192–201.

[7] S. C. Jardin, N. Ferraro, X. Luo, J. Chen, J. Breslau, K. E. Jansen, and M. S. Shephard, *The M3D-C1 approach to simulating 3D 2-fluid magnetohydrodynamics in magnetic fusion experiments*, J. Phys. Conf. Ser., 125 (2008), p. 012044.

[8] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392.

[9] J. Kim, W. J. Dally, S. Scott, and D. Abts, *Technology-driven, highly-scalable dragonfly topology*, in 2008 International Symposium on Computer Architecture, June 2008, pp. 77–88.

[10] W. Kohn and L. Sham, *Self-consistent equations including exchange and correlation effects*, Phys. Rev., 140 (1965), pp. A1133–A1138.

[11] L.-Q. Lee, Z. Li, C. Ng, and K. Ko, *Omega3P: a parallel finite-element eigenmode analysis code for accelerator cavities*, , Stanford Linear Accelerator Center (SLAC), 2009.

[12] X. S. Li, *Factorization-based sparse solvers and preconditioners*, in Series in Contemporary Applied Mathematics: Vol 19, Matrix Functions and Matrix Equations, Z. Bai, W. Gao, and Y. Su, eds., World Scientific, 2015, pp. 109–137.

[13] X. S. Li and J. W. Demmel, *Making sparse Gaussian elimination scalable by static pivoting*, in Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, SC '98, Washington, DC, USA, 1998, IEEE Computer Society, pp. 1–17.

[14] ———, *SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Trans. Math. Softw., 29 (2003), pp. 110–140.

[15] L. Lin, J. Lu, L. Ying, and W. E, *Adaptive local basis set for Kohn-Sham density functional theory in a discontinuous Galerkin framework I: Total energy calculation*, J. Comput. Phys., 231 (2012), pp. 2140–2154.

[16] W. Liu, A. Li, J. D. Hogg, I. S. Duff, and B. Vinter, *Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides*, Concurrency and Computation: Practice and Experience, 29 (2017), pp. e4244–n/a. e4244 cpe.4244.

[17] J. Mayer, *Parallel algorithms for solving linear systems with sparse triangular matrices*, Computing, 86 (2009), p. 291.

[18] P. Raghavan, *Efficient parallel sparse triangular solution using selective inversion*, Parallel Processing Letters, 08 (1998), pp. 29–40.

[19] F.-H. Rouet, *Memory and performance issues in parallel multifrontal factorization and triangular solutions with sparse right-hand sides*, theses, Université de Toulouse, Dec. 2012.

[20] Y. Saad, *Iterative methods for sparse linear systems*, Society for Industrial and Applied Mathematics, second ed., 2003.

[21] Y. Saad, *Numerical Methods for Large Eigenvalue Problems*, Society for Industrial and Applied Mathematics, Philadelphia, second ed., 2011.

[22] M. W. Sid Lakhdar, *Scaling the solution of large sparse linear systems using multifrontal methods on hybrid shared-distributed memory architectures*, theses, Ecole normale supérieure de lyon - ENS LYON, 2014.

[23] J. M. Soler, E. Artacho, J. D. Gale, A. García, J. Junquera, P. Ordejón, and D. Sánchez-Portal, *The SIESTA method for ab initio order-N materials simulation*, J. Phys.: Condens. Matter, 14 (2002), pp. 2745–2779.

[24] E. Totoni, M. T. Heath, and L. V. Kale, *Structure-adaptive parallel solution of sparse triangular linear systems*, Parallel Computing, 40 (2014), pp. 454 – 470.

[25] X. Wang, W. Liu, W. Xue, and L. Wu, *swSpTRSV: a fast sparse triangular solve with sparse level tile layout on Sunway architectures*, in Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2018, pp. 338–353.

[26] T. Wicky, E. Solomonik, and T. Hoefler, *Communication-avoiding parallel algorithms for solving triangular systems of linear equations*, in 2017

IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017, pp. 678–687.